Improving Integrated Operation in the Joint Integrated Mission Model (JIMM) and the Simulated Warfare Environment Data Transfer (SWEDAT) Protocol.

David W. Mutschler
NAVAIR Air Combat Environment Test and Evaluation Facility (ACETEF)
Code 5421, Bldg 2109, Suite 115
48150 Shaw Rd, Unit 5
Patuxent River, MD 20670
Email: david.mutschler@navy.mil
Phone: 301-342-6837
Fax: 301-342-6381

Abstract

The Simulated Warfare Environment Data Transfer (SWEDAT) is a shared memory interface currently managed by the Joint Integrated Mission Model (JIMM). It allows integrated operation of resources whereby the JIMM threat environment, stimulators, virtual cockpits, systems under test, and other agents are combined within the same simulation exercise. The Air Combat Environment Test and Evaluation Facility (ACETEF), the Joint Strike Fighter (JSF) Program, and other agencies use it extensively for both constructive analyses and real-time installed system test. Since its creation, JIMM and SWEDAT have been enhanced to improve capability and performance. More recent improvements include message queues, alternative coordinate systems, and dynamic simulated system control. This paper will describe the SWEDAT architecture, recent improvements, and planned efforts to further performance.

JIMM and SWEDAT

The Joint Integrated Mission Model (JIMM) is a general-purpose mission-level discrete-event simulator [Lat05]. The NAVAIR Air Combat Environment Test and Evaluation Facility (ACETEF), the Joint Strike Fighter (JSF) program, and many other efforts employ it for constructive analyses, training, and installed system test. JIMM is currently supported on Windows (2000 and XP), Linux, Silicon Graphics, and Solaris computer systems.

Systems in JIMM are explicitly represented. However, they are not modeled using detailed physics. Instead, each system operates in an “effects-based” manner. This simplifies internal calculations and permits thousands of platforms and component systems to interoperate while still maintaining real-time operation.

Thinking logic is controlled within JIMM players. Information used by this logic is based on perceptions. A perception is the specific data one player has about a platform. The only sources of this data are initialization, sensing, or communication. Hence, the perception’s data need not reflect “ground-truth” and can be both out of date and incorrect.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2005		**00-00-2005 to 00-00-2005**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Improving Integrated Operation in the Joint Integrated Mission Model (JIMM) and the Simulated Warfare Environment Data Transfer (SWEDAT) Protocol	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **NAVAIR Air Combat Environment Test and Evaluation Facility (ACETEF),Code 5421, Bldg 2109, Suite 115,48150 Shaw Rd, Unit 5,Patuxent River,MD,20670**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
Modeling and Simulation Conference, 2005 Dec 12-15, Las Cruces, NM

14. ABSTRACT
The Simulated Warfare Environment Data Transfer (SWEDAT) is a shared memory interface currently managed by the Joint Integrated Mission Model (JIMM). It allows integrated operation of resources whereby the JIMM threat environment, stimulators virtual cockpits, systems under test, and other agents are combined within the same simulation exercise. The Air Combat Environment Test and Evaluation Facility (ACETEF), the Joint Strike Fighter (JSF) Program, and other agencies use it extensively for both constructive analyses and real-time installed system test. Since its creation JIMM and SWEDAT have been enhanced to improve capability and performance. More recent improvements include message queues, alternative coordinate systems, and dynamic simulated system control. This paper will describe the SWEDAT architecture recent improvements, and planned efforts to further performance.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**16**	

A player also controls resources. These resources are encapsulated within systems that can be distributed onto different platforms at different locations. There are eight different system types: Sensor Receivers, Sensor Transmitters, Communication Receivers, Communication Transmitters, Weapons, Disruptors (Jammers), Movers, and Thinkers. A platform can have any number of systems with the exception that a platform can have no more than one mover system.

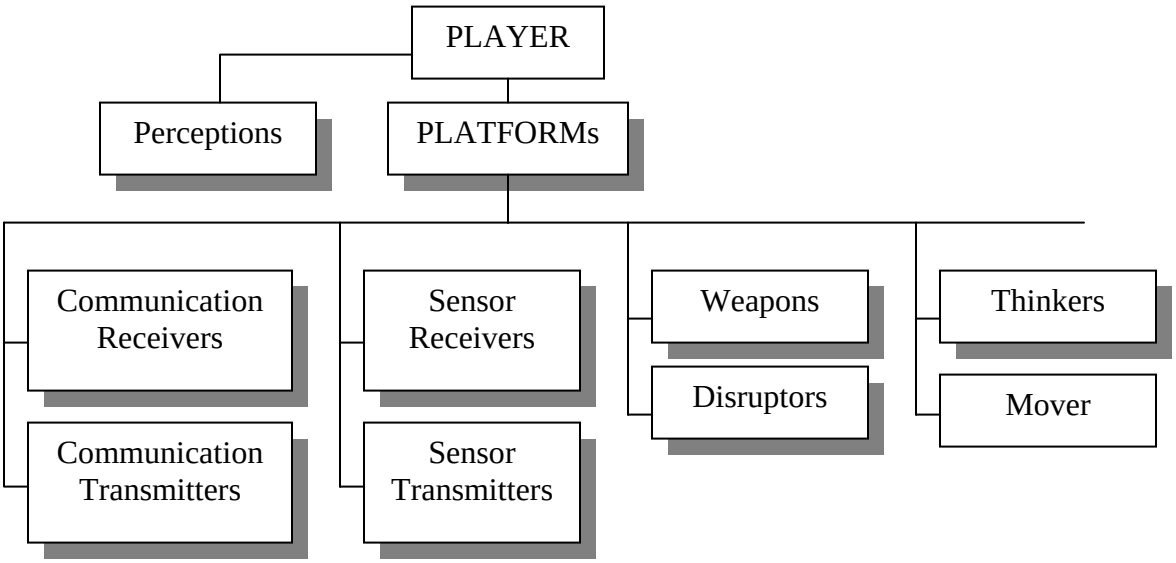

Figure 1 – JIMM Player Construction

JIMM works with other simulation tools via a shared-memory protocol known as Simulated Warfare Environment Data Transfer (SWEDAT). The shared memory can be either internal to the computer or can be distributed via reflective shared memory. In either case, processes assessing this shared memory (including JIMM) are known as "assets".

The SWEDAT protocol requires that one asset act as the "master model". The master model initializes the shared memory and configures it for use by the other assets. It also controls the memory and allocates it as needed to other assets. In this manner, the master model ensures that the integrity of the shared memory map is maintained. Normally, JIMM serves as the master model. It receives instructions about other assets via an input file known as the "Configuration Data Base" (CDB) and organizes the shared memory accordingly.

Some assets may passively view the JIMM simulation. However, JIMM allows other assets to assume control of specific systems. Hence, via interaction through SWEDAT, the asset can act and react as if it were working in the JIMM virtual environment. Also, an asset need not control all of the systems of a platform. One asset could control its weapons, another asset could control its mover system, and JIMM could control the remainder. Commonly, if the number of systems controlled is small compared to the remainder of the simulation and if those systems are part of a test or analysis, then JIMM

is also said to provide the “threat environment” and operate as the “threat environment generator”.

Some assets are standalone and interoperate with other assets solely via SWEDAT. However, many assets are actually interfaces. An interface (I/F) is a process that interacts with the shared memory and another process via some other protocol. This other process can be a viewer, a stimulator or another simulation. In addition, the other protocol can be a direct method such as a process specific shared memory or a distributed environment such as the Distributed Interoperable Simulation (DIS) or the High Level Architecture (HLA).

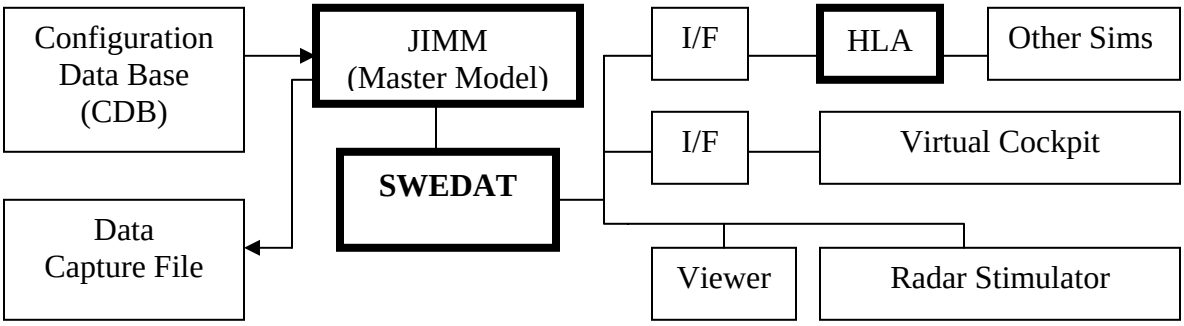

Figure 2 – Integrated Operation via SWEDAT

Multiple Coordinate Systems

SWEDAT currently supports two sets of coordinate systems. Each coordinate system is also supported by JIMM.

The first coordinate system is based on a flat tangential plane with a scenario center at some point on the earth. The ‘X’-coordinate roughly corresponds to ‘east’; the ‘Y’-coordinate roughly corresponds to ‘north’; and the ‘Z’-coordinate is up. By default, JIMM employs an orthogonal projection between the earth and the flat surface but can also be directed to use a transverse-mercator projection as well.

The other coordinate system places the origin at the center of the earth [TRW02b]. The ‘X’-coordinate intersects the earth at the equator and at zero degrees longitude; the ‘Y’-coordinate intersects the earth at the equator and the prime meridian; and the ‘Z’-coordinate in through the poles. This latter coordinate system is also known as “Earth-Centered Earth-Fixed” (ECEF). This coordinate system is assumed to be independent of the model of the earth (spherical or ellipsoidal).

Versions of SWEDAT

Within the JIMM community, there are some naming issues regarding SWEDAT. SWEDAT was once implemented using a version of shared memory known as “Multiport

Memory". Hence, the term 'MPM' had previously been used instead of SWEDAT and can still be seen in the JIMM instruction set.

In addition, SWEDAT is also known as the JIMM Shared Memory Interface (JSMI). This is how it is referenced in the JIMM Documentation. However, the term JSMI is no longer preferred by some individuals because of plans to allow other models and simulators to act as the "master model".

Lastly, there are currently three supported versions of the shared memory map used within SWEDAT.

1. A version with 32-bit floating-point precision that is backward compatible to all interfaces written for JIMM 2.2 and previous versions.
2. A version with 64-bit floating-point precision employed when greater accuracy is required such as when used the ECEF coordinate system. Members in SWEDAT data structures were rearranged to avoid problems with padding and to ensure alignment. This version was written for JIMM 2.3.
3. A version with 32-bit floating-point precision where the order of members in SWEDAT data structures match the 64-bit version.

During JIMM 2.3 and later, the term "SWEDAT" was used to specifically indicate the older version of the shared memory map. However, unless other specified, topics in this paper are pertinent to all supported versions of SWEDAT.

The JIMMLIB Library and Example Interfaces

In addition to the simulation model, the JIMM distribution also contains a 'C'-language library for use by asset interfaces. This library contains numerous procedures for purposes such as initialization, identifying controlled systems, finding asset specific information, SWEDAT memory access, procedures pertaining to the creation, access, and sending of dispatches.

The JIMM distribution also provides several dozen example interfaces. These include interfaces for DIS and HLA environments as well as assets specific to many commonly known asset types.

A current effort is to transparently expand the library by adding C++ language capability while allowing use of the same library for 'C'-language interfaces [Bal95]. This effort also includes software to automatically convert floating-point and integer representations to a common format as well as the ability to transmit SWEDAT information via a distributed network. Another effort to provide SWEDAT information via MPI has also been reported [Jones05].

Systems controlled by assets

Specific instructions about the functions in the systems controlled by assets are provided in JIMM through the Configuration Data Base (CDB). The CDB is a text file with instructions that include identification of assets, how they communicate, and initialization information.

Instructions specific to an existing player are roughly divided into three types: stimuli, decisions, and responses. Stimuli are dispatches from JIMM to the asset. Decisions are instructions as to which systems and which system functions the asset will control. Lastly, responses are dispatches from the asset back to JIMM.

In the following example (Figure 3), the asset will control some parts of the "vis_fighter_a/c" platform in the "71 vis_fighter" player. The specific platform must be identified since a player may have multiple platforms. From the STIMULI instructions, the asset will be informed whenever the fighter creates a new player and whenever it attempts to sense another player. From the DECISIONS, the asset will control movement (including when it crashes), weapon firing, and the changing of its signature (how well it is seen by other sensors). Other systems and functions will be controlled by JIMM. Lastly via RESPONSES, the asset will provide position and orientation (e.g. MANUEVER) information about the platform back to JIMM.

More recently, there has also been significant work in JIMM and SWEDAT to change the control of a system during a simulation run and also to dynamically change whether output is provided to an asset. This capability is also known as "dynamic asset allocation" [Mut02]. To assume control of a system (in whole or in part), an asset need only send a specific dispatch identifying the system and any specific system functions.

```
ASSET: 30 MFS
   EXISTING-PLAYER  71 vis_fighter  PLATFORM 1 vis_fighter_a/c
      STIMULI:
         CORRELATED-FIRING/BIRTH-ANNOUNCEMENT 3 BUFFERS
         PLATFORM-UNDER-TEST-IDENTITY $ only pertinent if
                                      $ SYSTEM-UNDER-TEST is set.
         SENSOR-CHANCE-STATUS FOR-THE SNR-RCVR 118 vis_optical-t_rx
         SENSOR-CHANCE-STATUS FOR-THE SNR-RCVR 119 infrared-x_rx
      DECISIONS:
         REACTIVE-MOVEMENT
         CRASH-CALCULATIONS
         LETHAL-ENGAGE-QUEUE-ADD
            THE 116 vis_dumb_bomb WEAPON
            FOR ALL TARGETS
         END LETHAL-ENGAGE-QUEUE-ADD
         LETHAL-ENGAGE-FIRING-START
            THE 116 vis_dumb_bomb WEAPON
            FOR ALL TARGETS
         END LETHAL-ENGAGE-FIRING-START
         DYNAMIC-SIGNATURE-CHANGES
            THE 15 vis_fighter_ele ELEMENT
         END DYNAMIC-SIGNATURE-CHANGES
      RESPONSES:
         MANEUVER
         DECISION-TO-FIRE 2 BUFFERS
            THE 116 vis_dumb_bomb
   END EXISTING-PLAYER
END ASSET
```

Figure 3 – CDB Instructions for a Single Player Asset

Position, Orientation, and State Information

Position and orientation (P&O) information is exchanged between JIMM and assets through specific blocks of shared memory. JIMM (acting as master model) creates a specific block in shared memory for each player, platform, and for the sensor systems, communication systems, disruptors, and weapon systems. Player and system blocks do contains some specific state information. However, since platforms are limited to one mover system, the position and orientation information is provided via the platform block.

When JIMM provides P&O information, it obtains the position, velocity, orientation, acceleration for a given simulation time. It then echoes this information (including the simulation time of the update) into the platform block. The asset samples this block and then uses the information as suits its needs. In a similar manner, when another asset controls the platform movement, the asset will provide the information and JIMM will sample and update its information on a periodic basis.

In the earliest versions of SWEDAT, there was an attempt to better ensure that the entire block of data corresponded to the specified time. In other words, the block would be written in an atomic fashion. A field was added to the platform block and initialized to zero. Assets reading the block would wait until the field was non-negative, increment the field count, read the data, and then decrement the count. Assets writing the block would wait until the field was zero, decrement the count, update the information, and then increment the count.

However, this attempt at coordination was abandoned. First, it did not fully provide mutual exclusion since there was still a possibility that one asset could be reading the data at the same time another asset was writing. Hence, the method's designation in the documentation is a misnomer. In addition, significant time was lost while one asset waited for another asset to free the block. A method using coroutines could have been implemented where 'ownership' of the block is transferred between assets. This would have solved the mutual exclusion problem but would not have solved the performance issues. In short, it was determined that the loss in performance was not worth as much as the assurance that the block corresponded to the specified update time - especially given the low probability that a reading asset and writing asset would access the block simultaneously. This probability is further reduced with the convention that the update time provided in the block is always the last data element modified. Lastly, in the rare case where that does occur, it is assumed that the update time is sufficiently frequent that the error of using 'older' data as part of an update is acceptable.

Even so, it is still assumed that the individual data items (i.e. integers and real numbers) are written atomically. This can be an issue in cases where the underlying hardware for reflective shared memory assumes that variables are no more than 32 bits in size and the use of the 64-bit shared memory map is desired.

Another issue that has arisen in JIMM is how "spread out" the update instructions should be. Initially, the update was executed for each instruction. However, if the instruction

pertained to a large number of platforms, then the time to execute the instruction could be large and this would cause periodic slowdowns in the performance of the simulation. To handle this problem, an event was created for each player. Furthermore, after initialization, the simulation time of the first update events was uniformly distributed given the update period. Hence, the processing of asset positions is now more evenly distributed.

P&O Updates from JIMM to Assets

Providing P&O information from JIMM to shared memory is a straightforward process. At simulation start, an event is created for each update. During event execution, the P&O information is obtained and echoed. A future event is then created given the defined update period.

Information should only be updated if it changes. If it doesn't change, then only the update time in the block is changed. This reduces performance overhead.

Another attempt to reduce cost was to allow the user to specify whether certain information was required. For example, JIMM will provide orientation as both forward and up vectors as well as roll, pitch, and yaw. If all assets in the exercise do not need to read the specific information, there is no need for JIMM to report it. Thereby, CDB instructions exist to inform JIMM not to provide the unneeded information. An example is provided below (where in JIMM, end of line comments are provided using the '$' character).

```
$
$ The following specifications determine how orientation of platforms
$ is sent to the assets from JIMM.
$
$     SEND-BOTH-VECTORS/ANGLES
      SEND-ROLL/PITCH/HEADING
```

Figure 4 – Instructions to Reduce Output Provided by JIMM

The period by which JIMM provides P&O information is also specified in the CDB. JIMM can also "smooth" its output of acceleration and orientation rates by averaging them with the values a small previous and the expected values a small time ahead. This modification of function is provided via the word "SMOOTHED" in the update instructions. An example of this instruction set is below. The "huge_bomber" and "sar_drone" are players in the scenario.

```
      UPDATE-MPM
         BY-TYPE    huge_bomber            EVERY  1.0 (SEC)
         BY-TYPE    sar_drone    SMOOTHED  EVERY  2.0 (SEC)
         ALL-OTHERS EVERY 2.0 (SEC)
      END UPDATE-MPM
```

Figure 5 – Update Instructions for Data from JIMM

A major issue with JIMM updates is the effect of interpolation between updates. In many cases (especially when the updates are several seconds apart), the position subsequently reported by JIMM may not correspond with the position dead reckoned by the asset. This

can be a problem especially in assets providing visual displays. A common phenomenon is for platforms to "jitter" on the display as they are updated.

Methods to limit the impact of jitter including "smoothing" the output given the different between the current displayed and the current reported position and orientation. In addition, the update rate can also be increased. For the future, plans exist to provide information pertaining to the next future path waypoint via a dispatch. Routines will be made available in JIMMLIB to interpolate the exact position in the JIMM simulation given a specific time.

Another problem manifests itself when the number of platforms in the scenario becomes large. Inspecting each every platform block and determining the reported location can be costly. However, a common method of avoiding this cost is to add a sensor to the platform specification. The sensor operates at the pertinent range and is programmed to only examine platforms of interest. A SENSOR-CHANGE-STATUS stimulus dispatch can then be sent to the asset. This dispatch will indicate when a platform becomes visible and when it is immediately out of range. Each for this immediate dispatch, no other dispatches occur when the platform is not visible. In this manner, JIMM can filter the platform information for the asset.

P&O Updates from Assets to JIMM

Reading P&O information from assets into JIMM has a set of issues that corresponds roughly with sending the data out from JIMM. However, one issue that is not addressed is when JIMM should consider the updated information. When provided by an asset, each P&O update for JIMM can be a costly operation. Therefore, it is highly desirable to minimize the updates as much as possible.

To minimize updates, JIMM currently provides thresholds for distance and angles. A distance threshold is the distance between the expected and reported positions given an update. In a similar manner, angle threshold is the difference between the expected orientation angles for roll, pitch, or yaw and the corresponding reported angles. If either threshold is exceeded, the JIMM updates the position and orientation within the simulation.

Distance threshold can also be specified as a percentage. In this case, JIMM divides the difference in distance by the distance between the location given at the last asset update and the current location. This value is then compared against the value in the CDB instructions.

```
MOTION-UPDATES-FROM-MPM
   BY-PLAYER  71 vis_fighter              EVERY  1.0 (SEC)
      USING POSITION-THRESHOLD 2.0 PERCENT
      AND ANGLE-THRESHOLD 1.5 DEG
END MOTION-UPDATES-FROM-MPM
```

Figure 6 – CDB Instructions for Updates from an Asset to JIMM

In handling updates from assets, JIMM does make some simplifications. First, it assumes that updates are point-to-point and not curved in arcs. In other words, the platform's path

can be expressed as a series of line segments. Furthermore, when JIMM receives an update, it avoids reaching the end of the platform's path by dead reckoning the path as a straight line to the edge of the scenario.

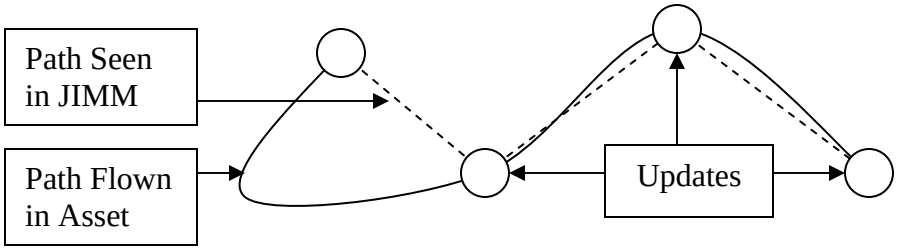

Figure 7 – Paths Generated Given Updates to JIMM

JIMM also assumes that the position reported by the asset is truthful and must be updated immediately. This causes a problem with "past history" since the path represented in JIMM will be updated and a location as previously noted for in a time in the past may not remain the same. This will have an effect on the scenario. For example, a sensing may have occurred on the projected path that would not have occurred on the path after it was adjusted. This effect is especially visible for platforms during turns and when the period between updates is sufficiently large. In most cases however, the expected error is assumed to be sufficiently small.

Thresholds can contribute to the scope of this error since the time of the last update was either the initial time or the time the threshold was last exceeded. Hence, the interval of time of altered "past history" could be large especially when platforms controlled by assets traveled in straight lines. However, modifications in JIMM to reduce the scope of the path change to the point where the last update would have occurred also reduces the magnitude of the potential error

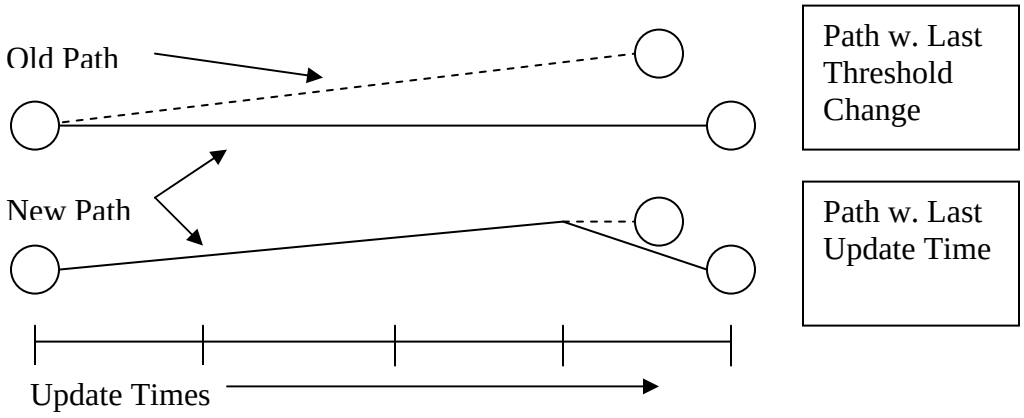

Figure 8 – Effects of Path Changes in JIMM due to Asset Position Updates

Another problem arose due to problems with different terrain maps. In some cases where different representations of terrain are used, an asset may assume a position is above the

ground where JIMM would assume it is below the ground. To handle this problem, an option was added so that an asset could "clamp" the platform to the ground and where reported altitude was assumed to "Above Ground Level" (AGL) as opposed to above the "Mean Sea Level" (MSL).

JIMM currently also has a limited capability to specify paths via the next waypoint. In this approach, JIMM will still control the movement of the platform and will provide specific location information via the platform block in the SWEDAT shared memory. However, the asset will specify the next path waypoint via a dispatch. When the dispatch is received, JIMM will adjust the path accordingly. This adjustment will include adjustments for turns, accelerations, and other factors.

JIMM Dispatches

In addition to position and orientation information, JIMM and assets can also exchange information via dispatches. Dispatches are provided via "mailboxes" that may be thought of as a unidirectional channel. Mailboxes are automatically established by JIMM (when acting as the master model) from itself to each other asset and also from each asset back to itself. Instructions also exist in the CDB to allow users to establish additional mailboxes between other pairs of assets.

JIMM Dispatches are communicated via two general methods: Dispatch Lists and Queues. The method used by a mailbox is specified in JIMM though the CDB instructions. The dispatch list method is the default.

Dispatch types are distinguished by a unique integer known as the action code and in general, have a defined size and structure. When instantiated, the shared memory associated with the dispatch is known as a "template". When a dispatch is to be read by the mailbox recipient, the action code is positive. When read, the action code is set to its negative inverse. This indicates that the associated memory may be reused.

Dispatch Lists

The dispatch list is the older method of asset communication via SWEDAT and many interfaces (including examples) still employ the method. A dispatch list is actually a list of templates. Receiving assets move from the head of the list to the tail of the list and inspect each of the action codes. If the action code is positive, it reads the information and reverses its sign.

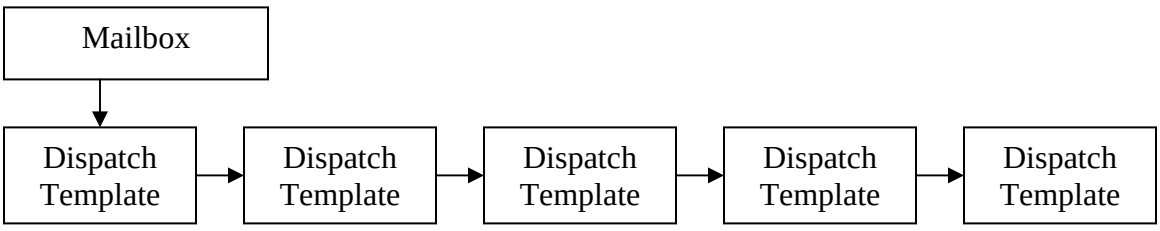

Figure 9 – Dispatch List Structure

When using dispatch lists, the dispatch templates must be created by JIMM during initialization or during runtime. A special dispatch template is provided on all dispatch lists for the request of additional templates. Assets search their list for this template, fill in the appropriate information, and then set the action code. JIMM will receive the message, create the template, and send a specific message in reply. This message will contain the address of the new message template.

The use of dispatch lists has a number of problems.

1. When there is a lot of communication, the number of templates the asset must continually inspect will be large and thus, can also have a large performance cost.
2. A dispatch that is only used once will have a performance impact throughout the course of the simulation run.
3. A large number of templates can consume a large amount of shared memory that cannot be safely reused.
4. The requirement to have JIMM create the template can introduce significant delays in interface processing.
5. There is no guarantee that dispatches will be received in the same order that they are sent.
6. Dispatches in mailboxes between different assets (other than JIMM) are restricted to using predefined templates.

There have some attempts to mitigate some of the problems. Some assets will restrict the number of templates inspected to a specified number. However, this introduces additional delays in dispatch delivery. Interfaces can also obtain templates in anticipation of later use. However, this will certainly impact space and may impact performance. Lastly, assets may impose their own ordering scheme and delay dispatch delivery until such ordering can be determined.

Dispatch Queues

Given the problems with dispatch lists, an alternative approach using circular queues was developed. A circular queue is an array. New dispatches are first written in the beginning of the array and subsequent dispatches are written in order. When the end of the array is reached, the dispatch will instead be written back at the array beginning. Dispatches are divided into separate blocks. These blocks are assumed to be contiguous and hence, if part of the block would exceed the array size, the whole block is instead written from the array beginning. By default, the size of the queue is set at 10240 integers but can be explicitly set by an asset.

In SWEDAT, queues also possess a “read” index controlled by the receiving asset and a “write” index controlled by the sending asset. The “read” index indicates the boundary of dispatch data read by the asset. In turn, the “write” index indicates the boundary of data “written” by the sending asset.

To send a message, the sending asset first determines the dispatch’s size and then using the “write” index, checks to see if the new dispatch would overwrite data not yet read by the receiving asset. If space in the queue is available, the sending asset then constructs a

template within the queue memory immediately after the position indicated by the write index. This template will have a negative action code. If space is not available, then the sending asset must postpone the sending of the dispatch, discard the dispatch, or otherwise handle it. Procedures are available in JIMMLIB to assist in creating the templates. Once the template is created, the "write" index of the queue is advanced to the end of the dispatch. The sending asset then fills in the information in the dispatch and lastly, sets the action code to its positive value.

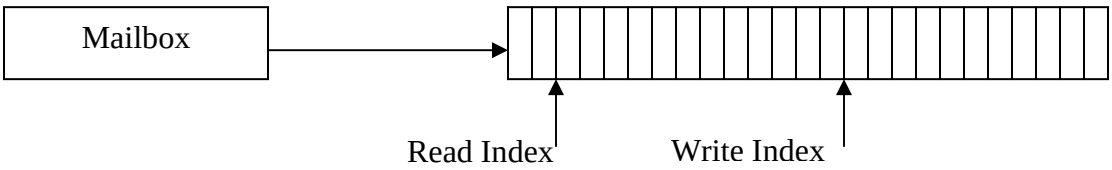

Figure 10 – Dispatch Queues

To receive a message, the receiving asset inspects the "read" index. If the "read" index is not equal to the "write" index, then a new dispatch has been created. The receiving asset then inspects the new dispatch's action code to see if it is positive. If so, then the receiving asset retrieves the information and then advances the "read" index to the end of the dispatch. Many assets will also set the dispatch's action code back to its negative value but this is not required when using queues.

Using queues avoids many of the problems with dispatch lists.

1. Comparing the "read" and "write" indices easily determines the availability of new templates.
2. Other dispatches can reutilized space used by read dispatches once the queue "wraps around".
3. The amount of shared memory dedicated to dispatches is fixed.
4. Assets create their own dispatch template without intervention by JIMM.
5. Messages will always be received in the same order they were sent.
6. Dispatches sent between different assets are not restricted to templates recognized by JIMM. In fact, they may use whatever template format they deem suitable.
7. Dispatches need not be a fixed size but only have to observe the limits inherent in the queue.

The implementation of queues however has not been without its difficulties. The main issue has been what to do when the queue fills up. In the initial implementation, JIMM would report the problem and then stop execution. However, this was not a satisfactory solution. Another implemented approach was to discard dispatches if a problem would result. However, this solution was not satisfactory because it gave older (and perhaps no longer necessary) dispatches a higher priority. Furthermore, there was no way to judge the relative importance of the dispatches lost versus dispatches retained. Another implemented approach was to double the size of the queue.

The last and most current approach is for JIMM to save dispatches on an "overflow" queue. This queue stows the dispatches in system memory in a first-in-first-out (FIFO) order. When memory in the queue is freed, dispatches from the overflow queue are written in. Newer dispatches are added to the overflow queue in cases where it is still not empty. In any case, should queue overflow be a problem, the recommended practice is still to increase the size of the queue if possible or ensure that the receiving assets reads its mail more frequently.

Dispatch Timing

With respect to timing, JIMM sends its dispatches as soon as they are generated. However, it will only inspect its incoming dispatches on a periodic basis or when it is waiting for wall clock time to "catch up" to its simulation game time. The time of this period between reading dispatches was initially fixed at 100 milliseconds. However, this period is now programmable via the CDB.

In addition, JIMM is currently restricted to reading one message per inspection. It is expected that this limit will be retained should dispatch lists be employed. However, in the newer versions of JIMM, dispatches are now transformed into events. This transformation requires much less immediate processing than handling the dispatches directly. Hence, coupled with the use of dispatch queues, multiple dispatches may more easily be processed within a single inspection.

Time Synchronization

During real-time operation, it was initially assumed that each asset could employ its own clock and that any skew that might result would be acceptable. If tighter synchronization were required, than assets would employ the time echoed by JIMM into SWEDAT. Hence, assets would not project system performance into a time later than the simulation game time.

Clocks in JIMM can be either external or internal. The internal clock is the system clock in the computer on which JIMM executes. The external clock is from another source and the time is then echoed by a utility into a specific location in SWEDAT. JIMM then coordinates the scenario using the echoed information as the time source.

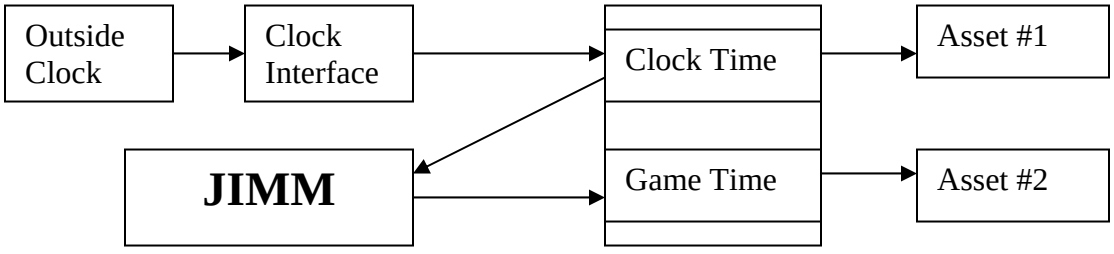

Figure 11 – External Time in JIMM

Other approaches have also been used for time synchronization. For example, a time-grant approach has been developed for use in HLA exercises [TRW03]. In another approach, JIMM will inspect the time reported by assets and only proceed if the difference between the current game time and the earliest reported game time is less than a specified interval [Mut05b].

Clock Flywheel Problem

One of the major issues that have arisen is how to handle cases where JIMM does not meet its real-time deadline. One simple method is for JIMM to work as fast as possible in processing events to have the simulation game time catch up to the wall clock. The problem is that should this catching up be required repeatedly, the simulation will slow down, rapidly speed up, slow down, and then rapidly speed up again. This is undesirable behavior for many assets – especially visual systems. This problem is otherwise known as the "flywheel" problem.

Another approach has been to cap the maximum speedup to a preset value. This allows the simulation to catch up smoothly and gradually. At one time, the maximum speedup was hard-code to 2.5%. However, the maximum time can now be programmed directly by the user [TRW02]. Unfortunately, this behavior is not ideal for assets where time synchronization is more information (as when interface send electronic stimulation to an aircraft during a test). In fact, the former behavior with the rapid speedup is preferred for some exercises.

Hence, since neither approach is best for all cases, the current implementation allows the user to determine the approach employed.

Conclusion

This paper has discussed the Joint Integrated Mission Model (JIMM) and the Simulated Warfare Environment Data Transfer (SWEDAT) protocol. Problems with dispatch mechanism, position and orientation updates, and clock synchronization have been discussed. Work to expand and improve SWEDAT operation continues.

Further information on JIMM and SWEDAT may be obtained from the JIMM Model Management Office (JMMO) at <jmmo@navy.mil>.

Acknowledgements

JIMM is the result of a lot of hard work by many people and many individuals contributed to the solutions discussed in this paper. The author apologizes for any omissions that may be made in these acknowledgements.

Peter Lattimore and others at Bosque Technologies originally developed the Simulated Warfare Environment Generator (SWEG). SWEDAT was originally implemented for SWEG and was carried forward when SWEG became the initial baseline for JIMM.

Jon Anderson provided the initial implementation for queues, the initial implementation for dynamic asset allocation, and the modification to update paths generated by updates from assets to the last update time.

Doug Pickeral provided the initial handling of queue overflows. Ross Jones of TRW developed the queue doubling solution. William Brooks provided the basic implementation of the overflow queue. Stuart Baldwin also implemented overflow buffers for SWEDAT. Blair Kitchen adapted this approach for the final implementation in JIMM.

The flywheel problem was identified and named by Phil Landweer. An option to initially switch between the gradual and rapid speedup approach was implemented by Jon Anderson. Ross Jones, Alex Zimmerman, and others from TRW (now Northrop Grumman Mission Systems (NGMS)) in Albuquerque NM provided the current solution using run speed control.

The author also acknowledges Ron Chesley and Ross Jones for their review and comments provided on drafts of this paper.

Current JMMO staff includes Natasha Bailey, Stuart Baldwin, Michael Chapman, Ron Chesley, Ralph Gibson, Alex Harper, Blair Kitchen, and Maritza Miller.

References

[Bal05] Baldwin, Stuart. "Shared Memory Interface Likability Engineering (SMILE)". JIMM Users Group, Rosslyn VA, May 2005. Available from the JMMO at <jmmo@navy.mil>

[Jones05] Jones, Ross. "JSMI Information via MPI", To be presented at the JIMM Users Group (JUG), Nov 2005. To be available via the JMMO at <jmmo@navy.mil>.

[Lat05] Lattimore, Peter et al. "JIMM Users Guide Volume One". Available via the JIMMO at <jmmo@navy.mil>

[Mut02] Mutschler, David. "JIMM Design Document for Dynamic Asset Allocation". 2002. Available via the JMMO at <jmmo@navy.mil>.

[Mut05b] Mutschler, David. "Master/Client Feasibility Study". Available via the JMMO at <jmmo@navy.mil>

[TRW02] TRW Systems Inc. "JIMM Design Document -- Run Speed Control". June, 2002. Available via the JMMO at <jmmo@navy.mil>

[TRW02b] TRW Systems Inc. "JIMM Design Document for Implementing the WGS-84 Earth Model". May 2002. Available via the JMMO at <jmmo@navy.mil>

[TRW03] TRW Systems Inc. "JIMM Design Document for Implementing Time Management, Revision 1". April 2003. Available via the JMMO at <jmmo@navy.mil>

Biography

The Naval Air Systems Command (NAVAIR) has employed David Mutschler in Modeling and Simulation (M&S) for over twenty years. He is currently working in the NAVAIR Battlespace Modeling and Simulation Division (Code 5.4.2) in Patuxent River, MD. Highlights in his career including acting as the Technical Lead for the Single Acoustic Signal Processor (SASP) Trainer (SAT) Upgrade for versions 4.0, 4.1, and 4.1.1 and acting as the Principal Investigator for "Parallelization of the Joint Integrated Mission Model Using Cautious Optimistic Control" for the High Performance Computing Modernization Program Office (HPCMPO). He received his doctorate in Computer and Information Science from Temple University in 1998. His current research interests include Parallel Discrete Event Simulation, Distributed Systems, Modeling and Simulation, and Software Engineering. He is also an Assistant Professor at the Florida Institute of Technology (FIT) School of Extended Graduate Studies (SEGS). He is a member of ITEA, SMS International, IEEE, IEEE/CS, and ACM, and ACM/SIGSIM.